VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Plošinová skákací hra v prostředí Internetu

Side Scroller Game over Internet

2015 Jakub Jarek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Jarek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Plošinová skákací hra v prostředí Internetu
Side Scroller Game over Internet

Zásady pro vypracování:

Cílem práce je vytvořit plošinovou skákací hru v prostředí Internetu v jazyce java, která umožní kooperativní hru více hráčů.

Hra bude umožňovat:
1. Hra proti jednoduché umělé inteligenci.
2. Hru více hráčů přes Internet.
3. Vytváření a editaci jednotlivých herních úrovní.
4. Program umožní připojení do současně vyvíjeného portálu her v jazyce Java.

Práce bude obsahovat:
1. Implementaci plošinové skákací hry.
2. Popis programátorského řešení s využitím diagramů jazyka UML.
3. Uživatelskou dokumentaci aplikace.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] DARWIN, Ian F. Java cookbook. 2nd ed. Sebastopol, CA: O'Reilly, c2004, xxiv, 829 p. ISBN 05-960-0701-9. Dostupné z: http://it-ebooks.info/book/2249/

Dále podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 7. května 2015 ..... Jakub Jauk .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015 ..... Jakub Jauk .....

Rád bych poděkoval všem, co mě podpořili při tvorbě práce, zejména pak mému vedoucímu práce Ing. Davidovi Ježkovi, Ph.D za jeho pomoc a dobré vedení.

Abstrakt

Tato práce popisuje tvorbu 2D plošinové hry v jazyce Java v prostředí internetu. Cílem je seznámit čtenáře s architekturou hry samotné, se zvolenými postupy a zajímavými koncepty využitelnými při tvorbě her tohoto typu. Řešené koncepty jsou následující : Interpolace mezi pozicemi v závislosti na čase, Predikce na straně klienta, Serverová rekonciliace, Kompenzace latence, řešení kolize pomocí metod AABB(Axis-Aligned Bounding Box) a SAT(Separation Axis Theorem), triangulace polygonů metodou ořezávání uší, tvorba pole viditelnosti pomocí vrhání paprsků a hledání cesty grafem pomocí A* algoritmu. Práce dále popisuje práci s použitými knihovnami LWJGL(Lightweight Java Game Library) a Kryo.

Klíčová slova: bakalářská práce, Java, plošinovka v prostředí internetu, 2D, server-klient, AABB, SAT, A*, triangulace polygonu, Serverová rekonciliace, Kompenzace latence, vrhání paprsků

Abstract

This thesis describes makings of 2D online side scroller written in Java language. The goal is to introduce readers to the architecture of the game itself and chosen methods and also to interesting and useful concepts when making a similar type of games. Discussed concepts include : Interpolation between positions depending on time, Client-side prediction, Lag compensation, collision using an AABB (Axis-Aligned Bounding Box) and an SAT(Separation Axis Theorem), triangulation of simple polygons by Ear Clipping method, field of view by Ray Casting and finding a way through a graph with an A* algorithm. The thesis also describes a work with LWJGL (Lightweight Java Game Library) and Kryo libraries.

Keywords: bachelor thesis, Java, online platformer, 2D, server-client, AABB, SAT, A*, polygon triangulation, Server reconcilliation, Lag compensation, Ray Casting

Seznam použitých zkratek a symbolů

IP	–	Internet Protocol
LWJGL	–	LightWeight Java Game Library
OpenAL	–	Open Audio Library
OpenGL	–	Open Graphics Library
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol

Obsah

1 Úvod

Zadáním bakalářské práce je vytvoření plošinové skákací hry v jazyce Java v prostředí internetu - bude umožňovat jak hru proti ostatním hráčům tak proti jednoduché umělé inteligenci. Hra by také měla mít jednoduchý editor úrovní a posledním bodem zadání je umožnění připojení do současně vyvíjeného portálu her - ten však prozatím neexistuje, tudíž se jím práce nebude zabývat. Téma bakalářské práce jsem si zvolil, protože mě tvorba her zajímá a ještě nikdy jsem žádnou hru po síti netvořil. Další důvod je ten, že by tato práce mohla být užitečná pro vytvoření představy o tvorbě podobných her. Stručná kapitola 2 udává základní představu o herních principech, podle kterých jsem volil vhodnou funkcionalitu jednotlivých částí aplikace. V kapitole 3 popisuji práci s knihovnou OpenGL a OpenAL, a funkcionalitu na těchto knihovnách postavenou. Principy popsané v kapitolách 4-6 jsou, stejně jako prakticky celá jejich implementace založeny na sérii článků [7] a článku [8]. V těchto kapitolách se zabývám zejména síťovou funkcionalitou a problémy s ní spojenou. V kapitole 7 popisuji knihovnu pohybu - techniky výpočtu kolize se světem a jejich použití. Kapitoly 8 a 9 spolu souvisí - kapitola 8 popisuje základní návrh editoru úrovní, přičemž kapitola 9 popisuje jednu z hlavních funkcionalit v editoru použitých - triangulaci kolizních polygonů entit vkládaných do světa. Kapitola 10 probírá A* algoritmus, a jeho využití pro umělou inteligenci. Následující kapitola 11 řeší tvorbu pole viditelnosti pomocí vrhání paprsků a využití tohoto konceptu v rámci herního návrhu. Kapitola 12 pak zahrnuje popis nalezení průsečíku dvou úseček(Dá se aplikovat i na přímky) - hledání tohoto průsečíku je možno využít při případné implementaci dvou v práci popisovaných algoritmů - při triangulaci polygonů (Kapitola 9) a při tvorbě pole viditelnosti (Kapitola 11). Vzhledem ke krátkému období, které jsem měl na vytvoření hry, jsem se při její tvorbě věnoval spíše zajímavým technikám použitých při tvorbě her tohoto typu a jejich implementací, než prototypováním herních ideí. Při vypracovávání práce jsem se snažil o určitou návaznost kapitol, na druhou stranu jsem kapitoly rozdělil na části, které by jako celky měly samy o sobě dávat smysl. Snažil jsem se rovněž nezacházet do přílišných detailů, a podávat pouze informace nutné při případném rozhodnutí čtenáře použít daný koncept či algoritmus.

2 Herní design

Herní principy jsou velmi jednoduché, a daly by se přibližně shrnout do následujících bodů :

- Hra je střílečka ve 2D prostoru probíhající v prostředí internetu.
- Hráči proti sobě, či proti umělé inteligenci(ta je blíže popsána v kapitole 10), mohou bojovat na mapách, které jsou vytvořeny pomocí jednoduchého editoru map(popsán v kapitole 8).
- Hráči je umožněno míření myší, střílení, skákání.
- Hráči nevidí přes statické kolizní objekty ostatní hráče(Blíže popsáno v kapitole 11).
- Hráči slyší ostatní hráče vzhledem k jejich pozici do určité vzdálenosti (skoky, střelba).

Podle výše zmíněných bodů se odvíjela prakticky celá práce co se týče konceptů nutných implementace, a proto je zde dále nebudu popisovat, a raději je popíši v kapitolách či sekcích jím náležícím.

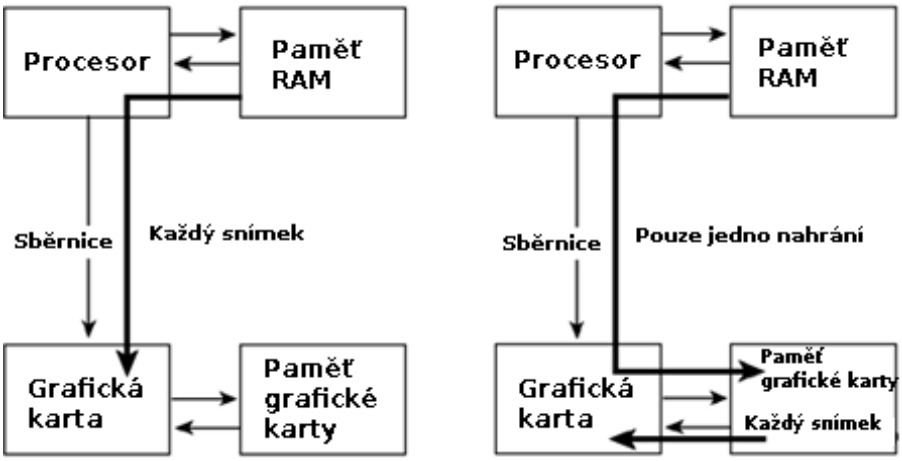
3 Použité knihovny

3.1 Úvod

Protože jsem už před psaním této práce a aplikace používal jak LWJGL[6], tak libGDX(knihovna na LWJGL postavená)[5], a vyhovovaly mi, nevyhledával jsem další možnosti. Obě z těchto knihoven umožňují jednoduchý přístup jak ke grafickému rozhraní (OpenGL), tak k rozhraní zvukovému (OpenAL). LWJGL však neobsahuje to, co tvoří z knihovny libGDX plnohodnotný 2D „engine“ - knihovny starající se např. o fyziku , či s ní spojené volitelné knihovny na libGDX postavené. Obě knihovny jsou multiplatformní, a umožňují jednoduchý přesun mezi podporovanými platformami. Zvolil jsem LWJGL, jelikož mi přišlo vhodnější použít „lehčí“ prostředí na typ projektu, který jsem se v tu chvíli chystal dělat - hru po síti.

3.2 Knihovna OpenGL

OpenGL není knihovna zaměřená pouze na jednoduché 2D aplikace jako je má hra : právě naopak - to, co umí nejnovější DirectX verze dokáže bezprostředně i nejnovější OpenGL verze. Zatímco DirectX knihovna je čistě knihovna pro operační systémy, či produkty Microsoft (např. konzole Xbox One), OpenGL je knihovna, která by se dala nazvat nezávislou na platformě - v podstatě se jedná o množinu funkcí, vytvářející jednoduchý přístup ke grafické kartě. Všechny výpočty OpenGL jsou založeny na multiplikacích různých matic. OpenGL nabízí různé typy vykreslování lišící se především způsobem zasílání dat do grafické karty. Ten nejjednodušší by se dal nazvat okamžitým vykreslováním - jedná se o vykreslování pomocí bloku mezi OpenGL funkcemi *glBegin* a *glEnd* - pro každé vykreslení se zasílají do grafické karty z paměti RAM znovu všechna data pro toto vykreslení potřebná a pro každé vykreslení bodu musí být volána minimálně jedna funkce [1] - je tedy jasné, že se nejedná o jeden z rychlejších způsobů - proto také není nadále oficiálně podporován. Je však zároveň nejjednodušší pro naprogramování, a kód nutný pro vykreslení bývá mnohem kratší než u ostatních metod. V aplikaci tento typ vykreslování stále používám pro jednoduché vykreslování čar. Pro vykreslování otexturovaných entit/polygonů jsem se rozhodl použít metodu VBO (*Vertex Buffer Object*). Ta spočívá v tom, že nejprve zašle všechna nutná data(pozice bodů polygonu, pozice bodů textury v lokálním prostoru) do paměti grafické karty, a následně s nimi pracuje pomocí číslicového (integer) identifikátoru. To znamená ohromné zrychlení zejména při vysokém počtu vykreslovaných objektů z důvodu ulehčení práce sběrnici mezi procesorem a grafickou kartou. Rozdíl mezi datovými pochody je znázorněn na obrázku 1 - tučně je znázorněn pohyb dat potřebných pro vykreslení.



Obrázek 1: Okamžité vykreslování(vlevo) a použití metody *Vertex Buffer Object*(vpravo) - obrázek převzat a přeložen z WWW : *http://www.yaldex.com/game-programming/0131020099_app02lev1sec10.html*

Po zaslání dat do grafické karty zvolí programátor tato data pomocí výše zmíněného identifikátoru a poté s nimi může dále pracovat. Pro změnu objektu před vykreslením se používají následující funkce : pro rotaci funkce *glRotate* a pro modifikaci pozice funkce *glTranslate*, pro modifikace velikosti pak funkce *glScale*, která . Důležitá vlastnost, kterou je nutné si uvědomit je, že OpenGL funguje jako stavový automat - pamatuje si tedy všechny doposud provedené změny. Proto existují OpenGL funkce *pushMatrix* a *popMatrix*. Funkce *pushMatrix* „uloží“ stávající stav koordinací OpenGL „světa“, funkce *popMatrix* tento stav „navrátí“. Další z věcí hodných zmínky je fakt, že jakákoliv transformace se neaplikuje pouze na jednu entitu - aplikuje se na celý svět - vykresluje se však pouze jeden z objektů - ten, na který byla tato transformace „směrována“. Pro podrobného průvodce moderním OpenGL bych doporučil navštívit webovou stránku „opengl-tutorial“ [2], či samotný web LWJGL [6].

3.2.1 Použití ve hře

Každý z objektů, který je možno vykreslit má dvě funkce - jednu pro inicializaci(ta se provádí pouze jednou - tvorba souřadnicových dat) a druhou pro vykreslení. Inicializace zahrnuje vytvoření polí se souřadnicemi jednotlivých bodů polygonu a koordinací textur a následné vložení těchto polí do objektů typu *FloatBuffer*. Dále se vygenerují samotné buffery díky OpenGL funkce *glGenBuffers*, jejíž výstup se uloží do např. globálního číslicového atributu typu integer. Toto integer číslo pak slouží pro identifikaci při přístupu k datům již zaslaných do grafické karty. Po proběhnutí této funkce může funkce druhá (funkce pro samotné vykreslování) tyto integer hodnoty používat pro zvolení dat, se kterými bude vykreslovací funkce pracovat (např. pomocí funkcí zmíněných v předešlém odstavci).

3.2.2 Textury a jejich alokace

Samotné použitelné objekty s texturou jsou tvořeny díky knihovně Slick2D [3], která je zároveň vybudována na knihovně LWJGL - textury se před jejich vykreslení pomocí funkce *glBindTexture* „nastaví“ jako aktuálně používané - není tak potřeba „nastavovat“ texturu znovu, pokud je následující objekt otexturován stejně. Textura je „nanesena“ v závislosti na geometrii objektu/polygonu a lokálními texturovými koordinacemi - obojí se nyní nachází v paměti grafické karty. I když je Java jazyk, ve kterém se o správu paměti v zásadě nemusí starat programátor (díky *Garbage Collectoru*, který „čistí“ paměť od alokovaných objektů, které již nejsou použitelné a přístupné programátorovi), stále je nutné ošetřit fakt, že je zbytečné jednu texturu nahrávat do paměti grafické karty pětkrát, když je ve hře pět hráčů, či jiných otexturovaných objektů. Zároveň by však textury neměly být statické v rámci jednoho objektu, protože nemusí všichni hráči vypadat stejně v jeden okamžik. Vytvořil jsem proto dvě třídy - *WorldTexture* a *TexturePool*. Jak již názvy napovídají - objekt třídy *WorldTexture* „obaluje“ samotný objekt s texturou(třída *Texture*), zároveň obsahuje informaci o cestě k této textuře. Pokud je potřeba určité textury, zavolá se pak statická funkce třídy *TexturePool* s cestou k této textuře - třída se v tuto chvíli podívá do listu již alokovaných textur - pokud najde hledanou texturu, navrátí ji a pokud ne, vytvoří ji a přidá do zmíněného listu a následně jí navrátí.

3.2.3 Animace

Animace jsem vyřešil velice jednoduše : textura je obalena do objektu *TextureWrapper*, který mimo samotné textury obsahuje i dvě časové hodnoty : „trvání“ textury a „uběhlý čas“ po použití textury. Pokud je tedy nutno vytvořit animaci např. chůze, lze objekty třídy *TextureWrapper* vložit do listu, který slouží jako animační smyčka pro danou animaci. Hráč si pamatuje index textury, kterou momentálně vykresluje, a modifikuje uběhlý čas textury časem uběhlým za daný snímek. Jakmile je dosažena doba „trvání“ textury, je „uběhlý čas“ momentálně vykreslované textury vynulován, a index v poli inkrementován o 1 (resp. navrácen na index 0 po dosažení konce).

3.3 Knihovna OpenAL

3.3.1 Použití knihovny OpenAL

Vzhledem k tomu, že již zmiňovaná mnou používaná knihovna LWJGL mimo OpenGL obsahuje i knihovnu OpenAL, což je multiplatformní 3D audio knihovna. Jelikož je zaměřena na zvuk ve 3D prostoru, rozhodl jsem se místo použití jednoho zvukového zdroje, který by se dal zpracovat pomocí OpenAL funkcí (ze kterých se mi žádaný efekt nedařilo dostat), použít zdroje dva, a vytvořit pomocí modifikace hlasitostí 1D (osa X) „panning“ efekt - rozložení zvuku mezi reproduktory vzhledem k pozici poslechu.

3.3.2 Výpočet pozice zvuku

Výpočet nutného výstupu pro indikaci pozice zvuku vůči hráčovi jsem vyřešil následovně : Za předpokladu, že je zvuk na pozici hráče, zvuk bude z obou reproduktorů hrát (při požadované hlasitosti zvuku 100%) na 50% (dohromady musí být hlasitost 100%, pokud mají být změny lineární v závislosti na vzdálenosti od zdroje) - při změně pozice zvuku vůči hráči(posluchači) se vůči sobě budou měnit i procentuální hlasitost daných zdrojů. Je tedy důležité určit, kolik horizontálně za sebou jdoucích pixelů znamená 1%, a podle toho hodnoty změnit. Rovněž jsem se rozhodl měnit procenta hlasitosti v závislosti na Eukleidovské vzdálenosti, aby se kromě horizontálního směru zvuku dala ze zvuku vyčíst vzdálenost od zdroje.

3.3.3 Dynamická změna zvuku

Protože je hra rychlá, může se změnit pozice zvuku vůči posluchači velice znatelně během krátkého časového úseku. Absence toho efektu se mi nelíbila, a tak jsem se rozhodl zvuky „obalovat“ do objektu třídy *SoundEffect*, obsahující časovač, který značí délku zvuku. Po dobu této délky je zvuk uložen v listu (spolu se všemi ostatními zvuky), a každý vykreslený snímek se pozice zvuku aktualizuje. Pokud tedy postava probíhá okolo momentálně skákajícího hráče, uslyší zvuk skutečně tak, jakoby se tak dělo. Po uplynutí času zvuku se daný objekt třídy *SoundEffect* odstraní z výše zmíněného listu.

4 Síťový návrh

4.1 Výběr typu sítě

Možností jak může komunikace mezi hráči probíhat je mnoho. Jedna z nich je klient-klient architektura, v jejíž případě zasílají všichni klienti data (svou pozici, koho zabili apod.) všem ostatním klientům v předem určených periodách. Zde vzniká problém - hráč může zasílat smyšlené pozice a další údaje, které jsou stěžejní pro správný chod hry. Tento problém řeší klient-server architektura. Funguje tak, že klient ztrácí autoritativitu nad světem - jediný správný stav světa drží server, a podle něj se všichni klienti řídí. Klienti zasílají pouze data o stisku kláves, pozici myši apod. Zároveň je chod dat centralizován, a je tak jednodušší implementovat administraci probíhající hry. Nevýhodou je však fakt, že hráč vždy po stisku klávesy musí počkat, dokud nedostane od serveru odpověď, ve které je po desítkách či dokonce stovkách milisekund jeho pozice. Vybral jsem si klient-server architekturu, na které jsem hru postavil, protože se jevila lepší volbou pro tento typ hry, a její negativa se dají poněkud snadno odstranit - na což se budu snažit navázat v následujících podsekcích a kapitolách.

4.2 UDP vs TCP/IP

Protože se jedná o hru po síti, jeden z hlavních problémů na který jsem při počátečním navrhování narazil bylo, pomocí kterého z protokolů transportní vrstvy (TCP či UDP)budu data posílat mezi hráči. Protokol TCP/IP sice zaručí doručení a správné pořadí paketů, to však dělá na úkor velikosti hlavičky samotného paketu. Dalším problémem je fakt, že odesilatel vždy čeká na potvrzovací paket, a posílá svůj paket znovu dokud potvrzovací paket neobdrží. Naproti tomu protokol UDP nezaručuje vůbec nic kromě toho, že paket vyšle na příslušnou IP adresu a port - hlavička je však podstatně menší, a na nic se neček á. To je velká výhoda např. u audio/video *streamů*, či akčních her - a také důvod, proč jsem si tento protokol zvolil.

4.2.1 Porovnání cest paketů

Jelikož je TCP/IP protokol spojově orientovaný, je nutné mezi serverem a každým z hráčů vytvořit spojení 1:1. U UDP to funguje jinak. Server nemusí mít 8 spojení 1:1 pro 8 hráčů - všechny datagram pakety může přijímat na jeden soket, a poté podle určitého identifikátoru data rozdělit do případných vláken starajících se o logiku a odesílání dat zpět klientovi.

4.3 Návrh strany klenta

Klient by měl umět komunikovat se serverem a nějakým způsobem zobrazovat výsledky této komunikace. Rozhodl jsem se pro následující řešení : Jedno vlákno bude svázáno s OpenGL(LWJGL) a bude zároveň sloužit k výpočtům, které prakticky nebudou mít žádný vliv na to, co se bude odesílat na server (výpočet interpolace, viditelnosti, stínů). Dále bude snímat klávesnici a myš. Druhé vlákno se bude starat o vytváření kopií snímků

klávesnice, tyto snímky bude zasílat na server frekvencí rovné serverové frekvenci, a zároveň tento snímek uloží do hráčova listu obsahujícího frontu pro predikci, které se věnuji v jedné z následujících podsekcí. Poslední vlákno se stará o příjem dat ze serveru a rozdělení těchto dat mezi entity světa (zahrnuje i funkcionalitu, kterou může mít „přijímací“ funkce - vč. řazení přijatých dat).

4.4 Návrh strany serveru

Server by měl přijímat vstup od klientů, vyhodnocovat tento vstup a posílat klientům zpět jejich pozici. Rozvržení je následující : vlákno objektu třídy Server se stará o příjem datagram paketů (pakety protokolu UDP), jejich deserializaci, a následné „rozeslání“ jednotlivým hráčům. O každého hráče se stará jedno vlákno, a to vlákno objektu třídy *ServerWorker*.

4.4.1 Vlákno objektu třídy ServerWorker

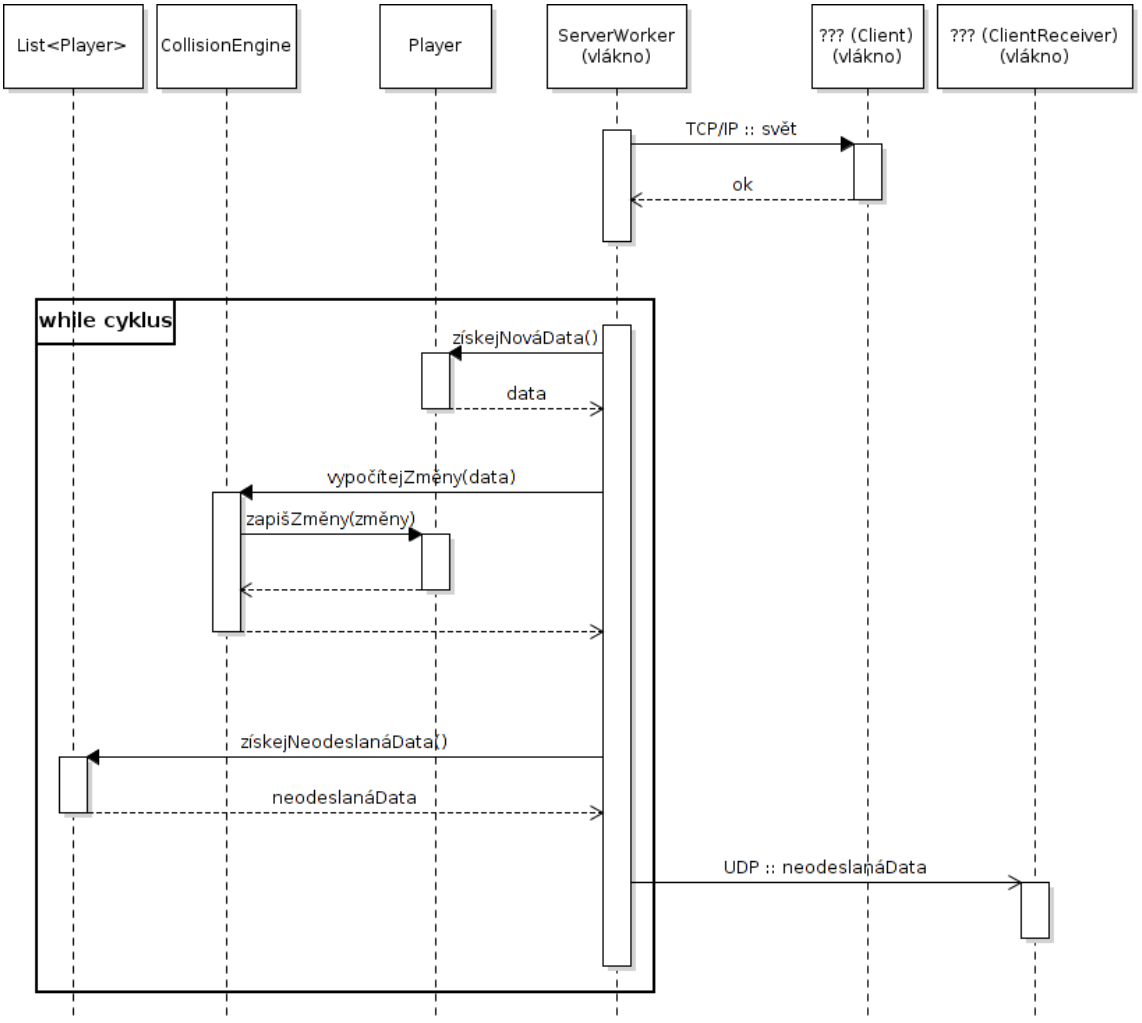
Jedná se o vlákno, které hráče kontroluje pro nové vstupy - jakmile je zjištěn nový vstup, vypočítá se na něj reakce - ta se zároveň uloží do listu hráče jako snímek. K těmto snímkům mají všechna vlákna ostatních hráčů přístup. Každé z vláken si tak uchovává záznam o tom, který snímek ostatních hráčů zaslal naposled - odesílá všechny doposud neodeslané informace. To na jednu stranu vytváří odchylku co se týče velikostí zasílaných paketů, na druhou stranu se však nečeká, dokud se všichni hráči „pohnou“. Sekvenční diagram přibližného chodu tohoto vlákna je zobrazen na obrázku 2.

4.5 Časový krok serveru a jeho podmnožina

Časový krok určuje časový úsek, za který proběhne jeden „výpočet světa“ serverem, a především frekvenci zasílání paketů mezi stanicemi. Rozhodl jsem se pro následující řešení : Na straně klienta snímám vstup klávesnice v předem určených dílech časových kroků serveru - např. časový krok : 60 milisekund, snímání : 20 milisekund. Všechny tyto snímky jsou zaslány dohromady. Čím větší frekvence snímání, tím je nejen větší přesnost pohybu, ale v případě závislosti predikce klienta(koncept blíže popsán v kapitole 5) na snímcích i vyšší responsivita.

4.6 Nepřesné probouzení vláken

Je všeobecně známo, že Java funkce *Thread.sleep(long ms)* je nepřesná - není zaručeno, že pokud se vlákno uspí na 10 milisekund, tak se probudí přesně za tuto dobu - spaní navíc není jediná věc, kterou vlákno musí dělat. Odchylka od požadované doby spánku může být dostatečně vysoká, aby negativně ovlivnila hratelnost. Častým řešením je zaznamenávat v každé iteraci dané smyčky čas, za který proběhla (včetně se spaním), a od toho odečíst optimální dobu spaní. Tím se získá časový přesah, který se odečte v další iteraci od doby spánku. Frekvence tak sice bude stále kolísat v krátkých časových úsecích, ale průměrně se bude frekvence přibližovat čím dál víc té požadované. Toto řešení jsem

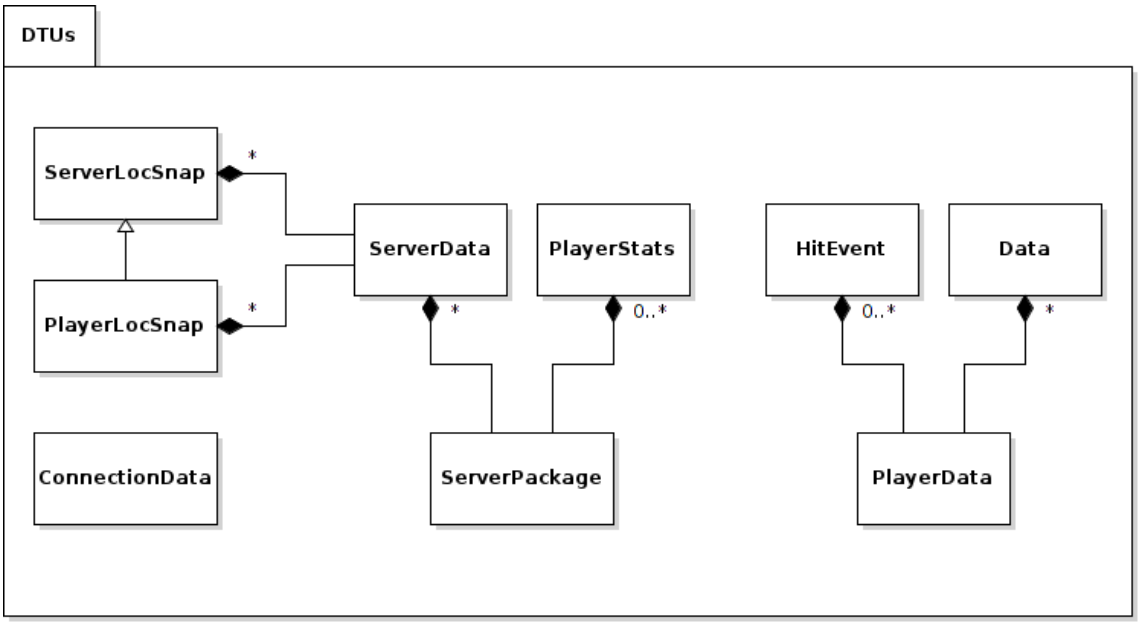


Obrázek 2: Zjednodušený sekvenční diagram chodu vlákna náležícího objektu třídy ServerWorker

použil pro všechny vlákna, u kterých to bylo nutné - vlákno třídy *Client* a vlákno třídy *ServerWorker*. Mimo tohoto řešení je také možné vyřešit tento problém modifikací délky samotných délek pohybů v závislosti na času od poslední přijaté pozice. Toto řešení jsem sice ze začátku používal, nicméně to znamenalo kolísající rychlost entit, a při pohybu více entit najednou bylo zrychlování a zpomalování jasně viditelné a nepříjemné.

4.7 Média přenosu

Již při testování prvních prototypů způsobu komunikace bylo jasné, že posílat celé serializované objekty (např. objekt se všemi informacemi o hráčovi na serveru) by bylo zbytečné, když potřebuje klient znát např. jen informaci o pozici, protože už dávno ví, jak je hráč vysoký, či co s onou informací dělat. Proto jsem vytvořil třídy s pouze důležitými informacemi, jejichž vzájemné vztahy jak na úrovni třídní, tak na úrovni objektové, jsou znázorněny třídním diagramem na následujícím obrázku 3.



Obrázek 3: Diagram tříd balíku **DTUs** (*Data Transfer Units* - Média přenosu)

Objekty třídy **ConnectionData** slouží pro navázání komunikace mezi klientem a serverem. Jsou serializovány, a následně zaslány pomocí protokolu UDP na jednotnou adresu serveru(adresa příjmu všech informací). Následně je vytvořeno vlákno objektu třídy **ServerWorker**, které jsem již probíral v předešlých podsekcích. Poté, co se zašle klientovi svět pomocí protokolu TCP/IP, používá toto vlákno objekty třídy **ServerPackage** pro zabalení všech potřebných informací pro klienta hráče - list objektů třídy **PlayerStats** obsahuje data o hráčích, která není nutno zasílat v každém „serverovém balíku“ - in-

formace o počtu zabití, smrtí apod. Dále „balík“ obsahuje list objektů třídy **ServerData**, které obsahují listy objektů **ServerLocSnap**, nebo **PlayerLocSnap**. Jak již názvy napovídají, **ServerLocSnap** obsahuje atributy, ve kterých jsou informace potřebné pro správný chod nepřátel(pouze pozice, životy, časovou známku, či objekty třídy **Data**, díky kterým je možno znát pokyny, na který je pohyb reakcí, či kam zrovna nepřítel míří). Tyto atributy dědí objekty třídy **PlayerLocSnap**, a přidávají k nim všechny ostatní nutné atributy nutné pro správný chod predikce na straně klienta(vliv gravitace, akcelerace, informace o předešlých stisknutých klávesách), kterou budu později probírat v kapitole 5. Co se týče zasílání dat ze strany klienta - stisknuté klávesy, časová známka, pozice mířidla jsou atributy třídy **Data**, jejíž objekty se vkládají do listu v objektu třídy **PlayerData**, a jsou zasílány společně s případnými záznamy o zásazích díky třídy **HitEvent**.

4.8 Serializace

Mým původním záměrem bylo využít serializaci a deserializaci objektů kterou umožňuje Java bez dalších potřebných knihoven - objekty tříd InputStream a OutputStream. Serializované objekty však byly překvapivě velké, i když v nich bylo pouze pár netransientních atributů. To se děje kvůli velikosti definicí jednotlivých tříd. Na internetu jsem tedy našel knihovnu Kryo [4], která tento problém řeší, a zároveň serializuje objekty mnohokrát rychleji - to samozřejmě závisí na mnoha faktorech. Při použití knihovny Kryo se obě strany (strana serializující a strana deserializující) musí domluvit na třídách, které se budou serializovat, a to ve stejném pořadí. Jednotlivé třídy se pak při serializaci a deserializaci identifikují pomocí integer čísla pořadí, ve kterém byla daná třída inicializována. Nevýhodou může být to, že třídy serializovaných objektů musí mít prázdný konstruktor. Třídy přitom nemusí implementovat rozhraní serializable.

4.8.1 Srovnání velikostí paketů

Následující tabulky porovnávají velikosti serializovaných dat posílaných ze serveru ke klientovi za provozu hry. V první tabulce byly velikosti měřeny při třech hrajících hráčích a v tabulce druhé při šesti hrajících hráčích.

	Minimální velikost dat [byte]	Maximální velikost dat [byte]
Java	1408	1724
Kryo	368	586
Zlepšení[%]	74	66

	Minimální velikost dat [byte]	Maximální velikost dat [byte]
Java	1919	2709
Kryo	641	1173
Zlepšení[%]	66	57

Rozdíly jsou na první pohled znatelné. Zpravidla procentuální zlepšení klesá se zvyšujícím se počtem záznamů - i co se týče rychlosti serializace. [15]

5 Principy strany klienta

5.1 Interpolace pozicí

Za předpokladu že bude hráč dostávat informace o své pozici jednou za 50 milisekund, a pozice by ve chvíli přijetí projevila, hrála by se hra jako při dvaceti snímcích za sekundu - což je nepřijatelné. Řešení je jednoduché : pokud znám dvě pozice a vím, jaký časový úsek je od sebe odděluje, můžu mezi nimi v závislosti na uběhlém čase lineárně interpolovat, čímž vytvořím plynulý pohyb mezi danými pozicemi. Vzorec vypadá následovně:

$$pozice = pozice1 + (t \cdot (pozice2 - pozice1)/t_s)$$

Uběhlý čas t je čas jednoho snímku(při 60 snímcích za sekundu to bude 16.667 milisekund apod.), přičemž t_s je čas, po který má tato změna pozic trvat. Interpolaci jsem tedy vyřešil následovně : vždy když od serveru přijme klient objekt s lokací hráče, přesune ho hráčovi do listu určenému pro interpolaci. Objekty mají nastavený časovač, který se při každém vykreslení obrazovky snižuje o čas, za který se vykreslení provedlo(čas snímku). Po vypršení času (Čas je nastaven na časový krok serveru, či jeho část rozdělení kroku na kroky menší - popsáno v minulé podsekci) je záznam odebrán. UDP datagram pakety však nikdy nebudou chodit přesně ve stejný čas, někdy mohou přijít ve špatném pořadí, někdy dokonce nemusí přijít vůbec. Proto jsem se rozhodl omezit interpolování objektů tímto způsobem - interpolace bude probíhat pouze pokud bude ve výše zmíněném listu dostatek záznamů pozic pro interpolování(předem rozhodnutá hodnota, např. 100 milisekund). To umožní při každé nově přijaté pozici tuto pozici zařadit na správné místo ve výše zmíněném listu pozic. Také to však znamená, že když po např. 100 milisekundách (např. v případě vzdálené komunikace) přijde odpověď od serveru se změnou pozice hráče, bude trvat dalších přibližně 100 milisekund, než se tato změna vůbec projeví. V tuto chvíli je hra v podstatě nehratelná.

5.2 Predikce na straně klienta

Jak již název napovídá, jedná se o „předvídání“ pozice hráče ještě předtím, než klient obdrží od serveru odpověď s pozicí. Klient počítá svět pomocí stejných „pravidel“ jako server - pohyb je tak responsivní, ale zároveň korektní podle určených pravidel. V ideálním případě by klient po obdržení „opravdové“ pozice nemusel nic dělat - tento případ může nastat však pouze tehdy, když ve světě nebudou ostatní hráči, nebo mezi hráči nebude interakce. Může nastat i problém ve floating-point rozdílech mezi jednotlivými výpočty z různých procesorů, či operačních systémů. Je tedy jasné, že tento mechanizmus na vše stačit nebude - v následující podsekci vysvětluji řešení tohoto problému. Je nutno podotknout, že pro počítání pozic ostatních hráčů tento koncept použitelný není - jsou tedy stále opoždění vůči hráčovi co se týče časových linií - problémy tímto vznikající a jejich řešení jsem popsal v kapitole 6.

5.3 Serverová rekonciliace

Serverová rekonciliace je zajímavý přístup k synchronizaci hráče se světem doplňující výše zmiňovanou predikci na straně klienta. Jakmile dorazí ze serveru odpověď na změnu pozice s korektní pozicí hráče, je vysoce pravděpodobné, že hráč bude už simulovat další změnu pozice, kterou server ještě nestačil zpracovat. Sama o sobě je nám pozice k ničemu - důležité je, na jaký dotaz (např. ve kterém čase - použití časové známky) je odpovědí. Klient tedy může jednoduše ukládat např. objekty s příkazy, které byly predikovány a v jakém čase se tomu tak stalo. Nyní stačí vrátit objekt (hráče) do času, který je totožný s tím z odpovědi od serveru, nastavit všechny potřebné atributy aby mu odpovídaly, a použít predikci na straně klienta na všechny zbývající příkazy, které poté jako jediné nadále zůstanou v „poolu uložených predikcí“, který je opět použit při dalším přijmutí dat od serveru. Pohyb je tedy naprosto responsivní, a když pozice kterou vypočítal klient nesedí s tou, jaká vyšla po výpočtech ze serveru, je možné buď to interpolovat jejich pozice, nebo hráče přesunout na pozici serverovou, což nevypadá vůbec dobře - jelikož jsem se rozhodl nepočítat kolize mezi hráči, nadále jsem se touto problematikou nezabýval, a pozice se při odchylkách jednoduše přepisují.

5.4 Částicové systémy

Přišlo mi vhodné použít částicové systémy, a to nejen k „oživení scény“, ale především k doplnění řešení kompenzace latence, které zároveň s tímto řešením popíšu v kapitole 6. Mé řešení částicových systému je následující:

- Částice je zobrazitelný objekt s volitelnou barvou, velikostí, který se pohybuje podle předem určených pravidel s určitou variabilitou - programátor určí např. minimální a maximální v určitých směrech či pozici, rotaci, nebo průběžné zvětšování - částice si pak při její tvorbě náhodně vybere např. pohyb a startovní lokaci mezi zadaným minimem a maximem.
- Částicový systém je objekt uchovávající a starající se o desítky až stovky částic - je možné přes něj startovat, restartovat na určitém místě(v případě uchování reference), či vypnout průběh částic.
- Částicové systémy lze jednoduše vytvořit a není třeba starat se o jejich mazání - po ukončení daného času jsou smazány z listu, ve kterém jsou všechny uchovávány, a ze kterého jsou zobrazovány.

Částicové systémy spouštím jako reakci na skok hráče, střelbu, zásah(ať už hráče, či jiné kolizní entity) s různými parametry. V případě střelby se např. spouští v místě, kde je přibližně hlaveň zbraně, a částečně přejímá rychlost projektilu ze zbraně letícího - opět však s určitou variabilitou.

6 Kompenzace latence

6.1 Nastínění problému

Hráč vidí svůj pohyb instantně, všechny ostatní hráče(dále protivníky) vidí s přibližnou časovou odchylkou *100 milisekund+odezva hráče/2+odezva protivníka/2*. Bez implementované kolize mezi klienty nyní prakticky hráč nedokáže rozeznat tuto hru od hry pro jednoho hráče - alespoň v rámci responsivity a hratelnosti. Běhání a skákání však není vše, co hráč bude muset umět. V případě této hry bude potřeba, aby uměl i střílet. Jak jsem již zmínil, všechny protivníky vidí hráč s určitým zpožděním, což můžeme vnímat jako určitou časovou linii. Hráč má svou vlastní linii se zpožděním 0 milisekund. Pokud tedy hráč zamíří na určitou pozici a vystřelí, míří a střílí na nepřítele, který na daném místě již nemusí být - tento fakt je vyobrazen na obrázku 4, v jehož případě nepřítel vyskočil, nicméně ve stejnou chvíli jako se on vidí ve vzduchu, ho jeho protihráč stále vidí na zemi - ač probíhá hra v lokální síti, rozdíly mezi pozicemi jsou příliš vysoké na to, aby byla hra hratelná. To je způsobeno automaticky řadící se frontou s pozicemi ze serveru, kterou jsem popsal v kapitole 5. Za předpokladu, že by hra neprobíhala po lokální síti je navíc třeba počítat s tím, že vzniká další zpoždění v závislosti na vzdálenosti mezi hráči a serverem. Hráčovi se tedy může běžně stát, že protivníka znatelně trefí, avšak v souběžně běžící linii serveru se tomu tak nestane. Protože je server autoritativní, klienti se jím musí řídit, a tak se zásah neprojeví.

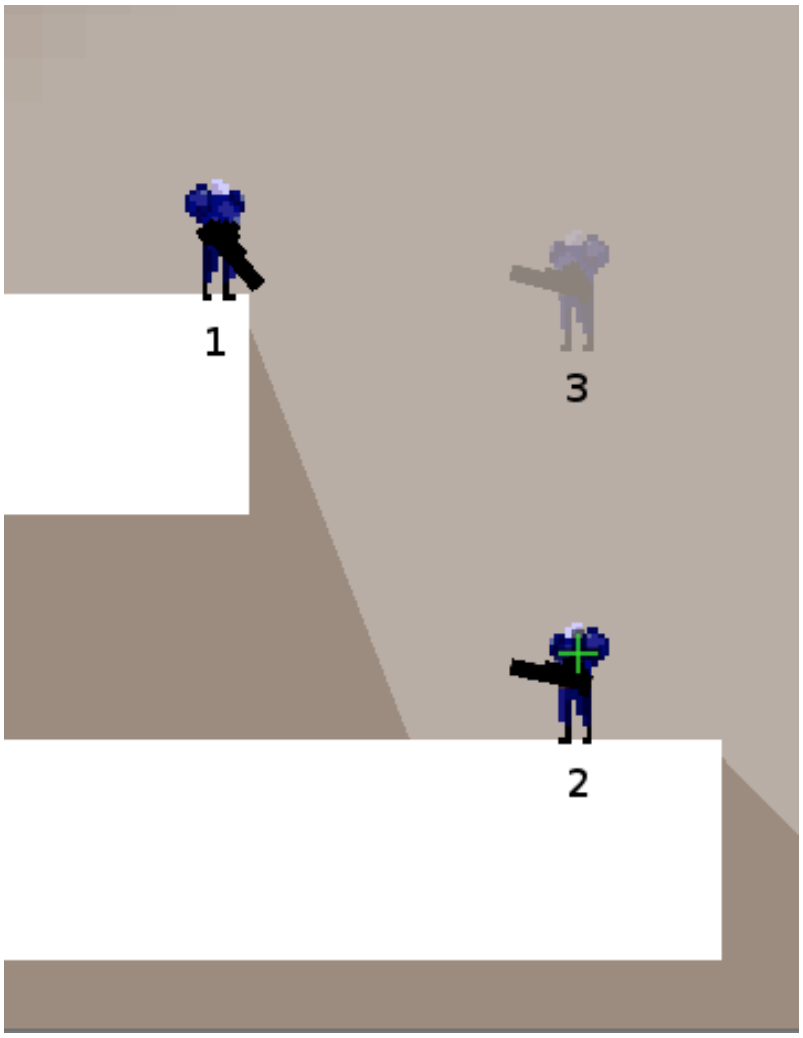
6.2 Řešení

Za předpokladu, že server dokáže zrekonstruovat stav světa v určité blízké minulosti (např. 2 sekundy), může klient posílat serveru informace o zásazích (čas zásahu, čas výstřelu, pozici míření apod.), server následně vypočítá, zda-li se zásah doopravdy stal - zjistí hráčovu pozici v čas výstřelu, protivníkovu pozici v čas zásahu, a vypočítá cestu projektilu podle hráčova míření a potřebných atributů hráče, které mohou mít určitý vliv na projektily (např. rychlost apod). Pokud tedy informace o zásahu nejsou smyšlené, zásah se projeví. To také znamená, že není potřeba simulovat pohyb pro případné projektily na straně serveru.

6.3 Doplnění řešení

Vzhledem k tomu, že používám viditelně pohybující-se projektily na místo jednoduššího řešení pomocí „neviditelných střel“, při střelbě nepřítele na pohybujícího se hráče může hráčovi připadat, že se střelám vyhnul, ve skutečnosti však byl nepřítelem zasažen. Ve hrách jako je Counter-Strike sice projektily vidět nejsou, tento jev se však stále projevuje např. při schovávání hráče za zeď - Hráč se může schovat za zeď, a následně umřít - zkrátka kvůli časové linii, na kterou nepřítel, který ho zabil reagoval. Tento problém se úplně vyřešit nedá, pokusil jsem se ho však zredukovat zrychlením projektilů a skrýt aplikováním částicových systémů - v případě zásahu hráčem se spustí ihned v místě zásahu, v případě zásahu hráče se spustí jakmile přijde ze serveru zpráva oznamující

jeho snížené zdraví. Hráč tím zároveň získává indikaci o tom, že byl zasažen a není tak vůbec nutné k tomuto zjištění vnímat samotné projektily - což zároveň pomáhá zakrýt výše zmíněný problém.



Obrázek 4: Porovnání časových linií při vnímání nepřítele(2) hráčem(1) na straně hráče, a nepřítelem na straně nepřítele(3) při skoku nepřítele

7 Knihovna pohybu

7.1 Představa

Jelikož by hra měla být plošinová skákačka, bylo potřeba rozmyslet se, jak se bude počítat pohyb, gravitace apod. Jedna z možností byla udělat si vlastní fyzikální/pohybovou knihovnu, druhá možnost byla použít některou ze známých a prověřených knihoven jako je např. box2D. Hra by rovněž měla probíhat v prostředí internetu, rozhodl jsem se tedy vytvořit si vlastní knihovnu - je důležité si zde uvědomit, že většina z funkcionality sofistikovanějších fyzikálních knihoven stejně nebude použita, a pouze by byla na obtíž (zejména při snaze co nejvíce zminimalizovat velikost posílaných paketů).

7.2 AABB kolize

Axis-Aligned Bounding Box (dále AABB) je jednoduchý koncept, na kterém měla původně celý systém kolizí světa hry stavět (stále je však důležitý). Za předpokladu že je svět postaven z obdélníkových objektů, jejíchž strany jsou rovnoběžné s osami x a y, rozděluje se problém do následujících podmínek :

- Je pravá strana objektu A vlevo od levé strany objektu B ?
- Je levá strana objektu A vpravo od pravé strany objektu B ?
- Je vrchní strana objektu A pod spodní stranou objektu B ?
- Je spodní strana objektu A nad vrchní stranou objektu B ?

Pokud platí, že kterákoliv z výše uvedených podmínek je pravda, znamená to, že objekty spolu nejsou v kolizi. Stranou je myšlen jakýkoliv bod na určité straně : levá strana je u objektu identifikována hodnotou atributu x, pravá strana hodnotou atributu x+šířkou objektu. Obdobně to funguje pro vrchní a spodní strany objektu : y je strana vrchní, y+výška objektu strana spodní. [11]

7.2.1 Možné aplikace AABB

Způsobů použití je opět více. Jeden z nich může být tento : pohneme s entitou předtím, než se otestuje, jestli bude s ostatními objekty v kolizi, a následně při případné kolizi ošetříme tuto situaci „vytlačením“ entity mimo kolidující objekt. Další možnost je otestovat před samotným pohybem případnou kolizi, která by při něm nastala, a podle toho se rozhodnout, zda s entitou pohnout či ne.

7.2.2 Složitost AABB kolize, optimalizace

V případě, že bych plánoval ve hře používat tisíce objektů v jeden okamžik, bylo by rozumné rozdělit svět do několika částí - kontrolovala by se pouze část, ve které hráč je. Mým plánem však bylo dělat světy, či herní mapy, které by objektů neměly tisíce, ale pouze desítky. Z toho důvodu jsem se tímto problémem dále nezabýval.

7.3 Složitější kolizní objekty

Čtverce a obdélníky sice zcela jistě stačí k vytvoření smysluplné hry - ve spoustě směrech je to výhoda (umělá inteligence, jednodušší náhodné generování obsahu), má představa však byla jiná - chtěl jsem mít možnost vytvoření jakéhokoliv tvaru, což by umožnilo tvorbu zajímavějších a detailnějších negenerovaných map. První řešení, které mě napadlo byla Per-Pixel kolize, kde by se použil alfa kanál textury, a zjišťovalo by se, kde se případné objekty ve světě překrývají podle porovnání hodnot daných pixelů (či větších fragmentů). Toto řešení jsem se však rozhodl nepoužít z důvodu toho, že jsem nechtěl, aby objekty musely mít kolizi rovnající se tvaru viditelných pixelů textury 1:1. Další možné řešení je použít SAT (*Separation Axis Theorem*), pro který jsem se rozhodl, a kterému se budu věnovat v následujících podsekcích.

7.4 SAT detekce kolize

Separation Axis Theorem (dále SAT) říká, že dva konvexní polygony spolu nejsou v kolizi pokud existuje osa, na kterou když polygony promítneme, nebudou se jejich projekce překrývat. Osa je normál strany, na kterou algoritmus momentálně testuje (osa je v pravém úhlu vůči testované straně) - to lze docílit např. změnou znaménka u Y souřadnice koncového bodu strany tvořené úsečkou. Z osy se vytvoří normalizovaný (jednotkový) vektor, a to následujícím způsobem : Nejdříve se spočítá délka vektoru, která se v případě 2D prostoru počítá pomocí následujícího vzorce :

$$\sqrt{(x \cdot x + y \cdot y)}$$

Poté, co je známa délka vektoru, vydělí se jí každá ze souřadnic (x a y v tomto případě) - tím je získán hledaný vektor délky 1. Samotná normalizace není nutná pokud by bylo potřeba pouze zjistit zda jsou objekty v kolizi. Bude však potřeba v jednom z následujících kroků. Pokud je vypočítán skalární součin určitého bodu (vnímaného jako vektor s počátkem v bodě 0) s vektorem určujícím osu, je získán skalár, který lze vnímat jako projekci - je třeba si přitom pamatovat, odkud-kam jsou dané projekce, což následně použít pro zjištění, zda se překrývají. Skalární součin se počítá následujícím způsobem :

$$(V1x * V2x) + (V1y * V2y)$$

Testují se takto všechny strany, dokud se projekce překrývají. Po první nalezené ose, na které spolu projekce nejsou v kolizi můžeme přestat počítat, protože máme výsledek - objekty spolu nejsou v kolizi. Pořád by se však objekty byly schopny pohybovat v něčem, co by se dalo představit jako jakousi mřížku možností pohybu. To znamená, že pokud např. půjde hráčova postava z kopce, bude to vypadat, jakoby šla po schodech. To na jednu stranu jde ovlivnit „citlivostí“ (v závislosti na časovém kroku serveru) pohybu, ale bez ošetření to zkrátka nikdy nebude vypadat dobře. Existuje jednoduché řešení : využít os které se již byly testovány pro zjištění, na které je nejmenší překrytí daných projekcí. Pokud je známo překrytí a původně testovaná osa je v jednotkovém vektorovém tvaru,

dá se vzájemným vynásobením získat vektor, který se nazývá minimální vektor posunutí. Ten se pak může použít jako odchylka hráčova pohybu (v případě že se „vyšší vrstva“ aplikace pro to rozhodne). Dva z kroků algoritmu při testování kolize dvou trojstranných polygonů pomocí SAT jsou znázorněny na obrázcích 5 a 6 (V případě vyobrazeném na těchto obrázcích jsou to zároveň jediné dva potřebné kroky).[9][10]

7.4.1 Konkávní polygony

Jak již bylo zmíněno, SAT se dá použít pouze pro konvexní polygony. Každý konkávní polygon se však dá rozdělit na množinu konvexních polygonů použitelných pro tuto metodu. Touto dekompozicí se budu víc do hloubky věnovat v kapitole 9 o triangulaci polygonů.

7.4.2 Složitost SAT kolize, optimalizace

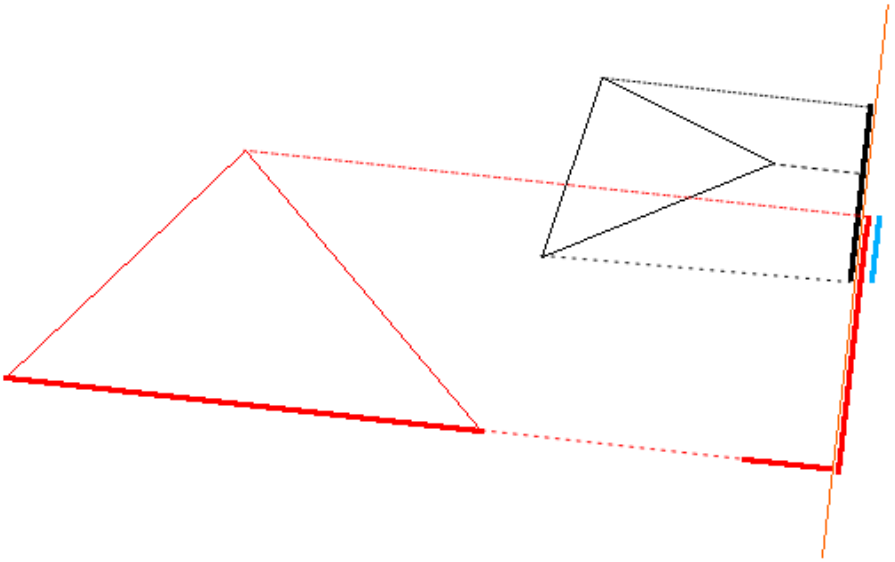
Výpočet kolize pomocí SAT je obecně rychlý, i tak je ale jeho výpočet mnohem náročnější než AABB. Velice jednoduše se však dá využít toho, že polygony jsou vždy uvnitř obdélníku určujícího „bounding box“. Při testování kolize tedy můžeme nejprve otestovat objekty pomocí AABB - pokud nejsou v kolizi podle AABB, nebudou v kolizi ani podle SAT, takže se SAT vůbec nemusí počítat.

7.5 Implementace pohybu

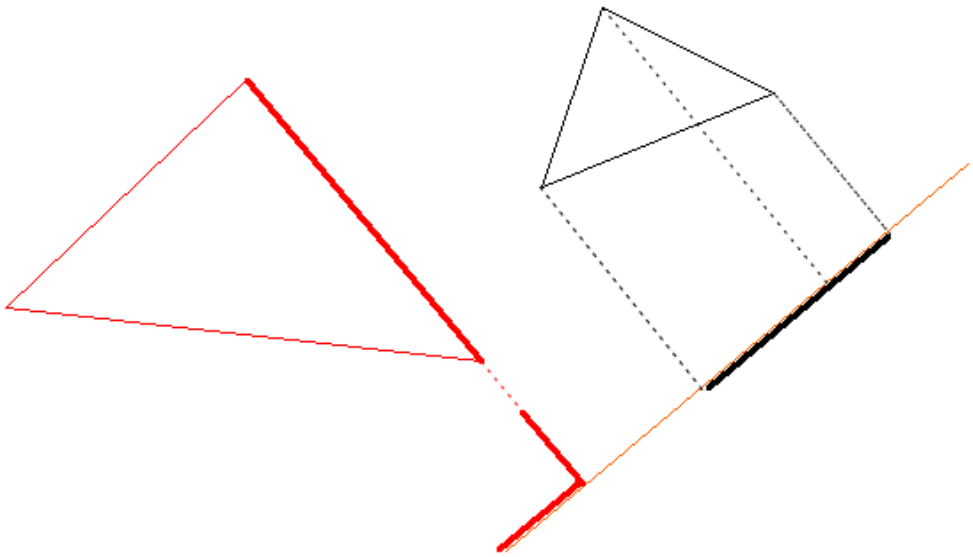
Samotný pohyb je řešen pomocí jedné statické funkce (pro server i klient), ve které se při výpočtech bere v potaz časový krok serveru pro jednotnost výpočtů. To znamená, že by se pohyb na straně klienta měl počítat ve stejných intervalech jako na straně serveru. Tento fakt velmi pomáhá při implementaci již zmíněné predikce pohybu v kapitole 5. Hráči se postupně zrychlují při pohybu určitým směrem a gravitace rovněž roste, pokud hráč nestojí na zemi. To se v mém případě děje jednou za každých 20 milisekund, což také znamená, že maximální použitá hodnota serverového časového kroku je 20 milisekund a měla by být taková, aby byla dělitelná zvolenou dobou části časového kroku pro jednoduchost použití.

7.5.1 Význam SAT pro pohyb hráčů

Za předpokladu, že se při každé kolizi připočítává již zmíněný minimální vektor posunutí při kolizi s objektem či polygonem, který nemá všechny strany vodorovné k osám x či y, vzniká problém - objekt „klouže“. Tento problém jsem odstranil tak, že pokud se hráč neníní pohybovat, vypíná se na tento čas přičítání minimálního vektoru posunutí, a dopočítává se pouze y souřadnice do potřebné pozice, kdy objekty nejsou v kolizi. Poté, co je však problém vyřešen, vytváří přičítání výše zmíněného vektoru pohyb, který je pomalý při chození do kopce a rychlý při chození z kopce - další výpočty nejsou potřeba (pokud není cílem reálnost pohybu a fyzikální přesnost - což v případě této hry není). Vliv úhlu na rychlost lze regulovat nastavením gravitace.



Obrázek 5: zjištěna kolize - modrou barvou je znázorněn případný minimální vektor posunutí



Obrázek 6: kolize nebyla zjištěna - pro tento polygon již nemusíme testovat

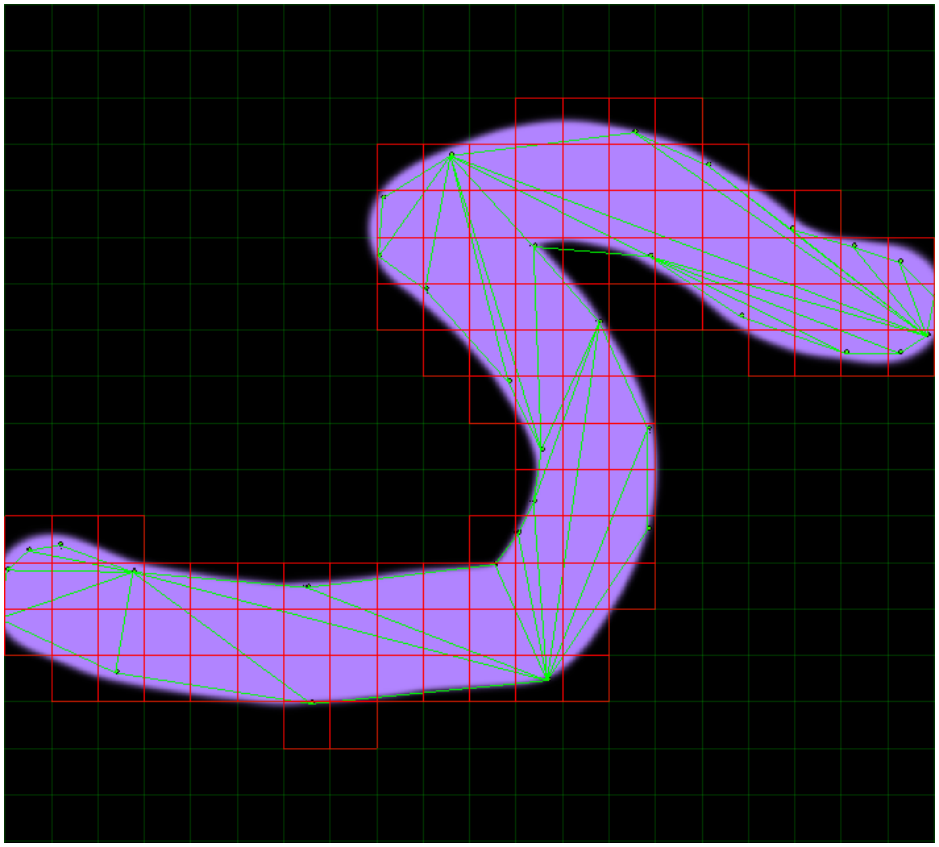
8 Editor map

8.1 Popis základních principů editoru

Hlavní ideou editoru map je vytvořit prostředí, které umožní co nejjednodušší vytváření a editaci úrovní. To mi poněkud komplikují některé koncepty popsané v předešlých kapitolách. Jeden z nich je SAT (*Separation-Axis Theorem*), který jsem zmiňoval v kapitole o knihovně pohybu - aby detekce kolizí fungovala jak má, musí být polygony, se kterýma se počítá konvexní. To se dá vyřešit buď to omezením herních kolizních objektů na pouze konvexní polygony, a nebo rozdělením konkávního polygonu na několik konvexních polygonů (např. trojstranných), což je jeden z důležitějších principů, který v této kapitole budu řešit. Nejdříve je však důležité, abych určil, jak prostředí funguje, a co vše umožňuje. To lze popsat v bodech, z nichž některé proberu více do hloubky v pozdějších podsekcích:

- Možnost importu obrázků ve tvaru png či jpg, v rozměrech, jejichž hodnoty jsou násobky čísla 2 (kvůli OpenGL) jako objektů, které se následně mohou vkládat do světa - lze zvolit velikost objektu, či se rozhodnout pro určitý násobek původní velikosti (libovolné floating-point číslo). Zároveň je možné zvolit si Z-pozici při vykreslování na obrazovku (a tím vrstvit objekty).
- Všechny takto vytvořené objekty se vkládají do objektu typu ArrayList, který je zobrazen pomocí tříd *JFrame* a *JLabel* - lze selektovat objekty, mazat je, apod.
- Při zvolení objektu z nabídky je možné ho vložit do světa - nyní nemá objekt nastavenou kolizi.
- Do světa lze vložit kolizní body, které se automaticky propojují v rámci objektů určených ke kolizi, a tvoří tím obrys objektu, což lze prozatím vnímat jako kolizi daného objektu.
- Objekty ve světě jsou propojeny pomocí identifikátoru s objekty v nabídce objektů - např. při přidání kolizních bodů se změny projeví na všech objektech pocházejících ze stejného objektu v této nabídce.
- Kolizní polygon se automaticky rozkládá na trojstranné polygony - tyto polygony jsou vyobrazeny na obrázku 7 uvnitř růžového objektu zelenou barvou (problematika je podrobněji popsána v následující kapitole).
- Objekty jde před vložením do světa horizontálně otáčet.
- Kolize objektu má vliv na uzly mřížky (pokud je s uzlem v kolizi, nastaví ho na uzel kolizní - červené uzly na obrázku 7) - touto mřížkou se řídí umělá inteligence, kterou blíže popisuji v kapitole 10.

- Další možnosti editoru jsou : vkládání jednoduché dynamické vegetace (náhodná velikost apod.), kolizní objekty určené pro umělou inteligenci (pro omezení pohybu), míst, na kterých se může hráč po smrti objevit, či bodů, které určují zajímavá místa pro hráče ovládané umělou inteligencí, které budu věnovat celou kapitolu 10.



Obrázek 7: Ukázka tvorby úrovně v editoru map

8.2 Serializace a ukládání

Objekty jsou serializované pomocí objektů typu InputStream a OutputStream (Java serializace) - ač jsem změnil způsob serializace při komunikaci mezi hráči a serverem, rozhodl jsem se nadále používat tyto třídy místo knihovny Kryo[4], protože by změna prakticky neměla vliv na chod hry. Po vytvoření dané mapy lze mapu exportovat - tato mapa si pak může uživatel zvolit při startu serveru, a každému hráči se při jeho připojení následně tato mapa zašle (všichni hráči by tedy měli mít potřebné textury - to jsem vyřešil neumožněním využívání textur mimo samotný projekt). Mapa se také dá uložit jako „projekt“, který obsahuje danou mapu a všechny v ní použité entity pro následné jednoduché upravování.

9 Triangulace polygonů - Metoda ořezávání uší

9.1 Úvod

Je mnoho možností, jak konkávní polygon rozdělit na množinu polygonů konvexních. Jednou z nejjednodušších jak na pochopení tak na implementaci je metoda ořezávání uší (*Ear clipping method*). Také je však jedna z pomalejších metod - což není příliš relevantní, protože se polygony nemají rozdělovat za hry, ale přímo v editoru úrovní. Přes jednoduchost tohoto algoritmu jsem při jeho implementaci narazil na spoustu problémů, jejichž řešení se v této kapitole budu snažit stručně a přitom co nejlépe popsat.

9.2 Princip metody ořezávání uší

Platí, že každý jednoduchý polygon skládající se s alespoň 4 bodů má alespoň 2 uši (zobrazeno na obrázku 8), což jsou trojstranné polygony, jejichž dvě strany jsou již spojené strany „obrysu“ polygonu, a třetí stranu tvoří strana, která spojuje dvě již známé strany, je celá uvnitř triangulovaného polygonu. [16] Pokud se tedy sekvenčně prochází body tvořící tento polygon, a je nalezeno zmíněné ucho, je ho možno oddělit od původního polygonu, a následně provést daný algoritmus znova s ořezaným polygonem - tento proces se opakuje dokud není triangulace ukončena.

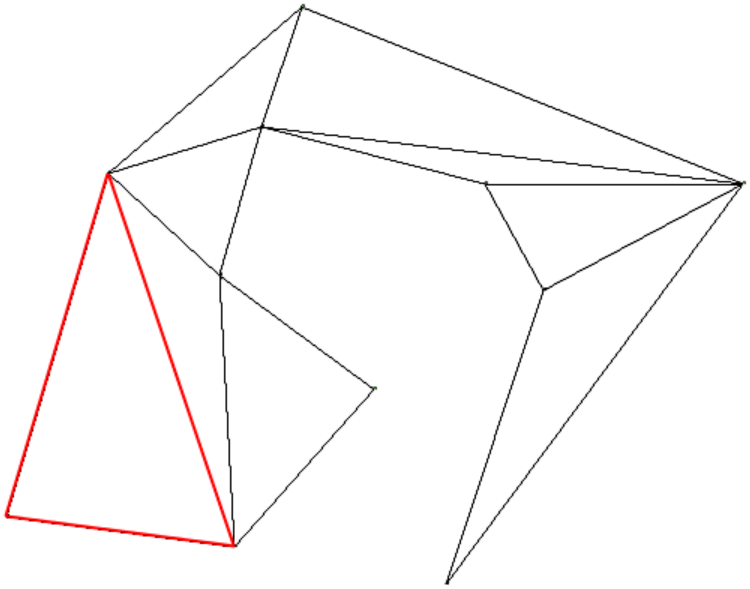
9.3 Popis řešení

Hlavním problémem, který jsem při tomto algoritmu musel vyřešit je samotné zjištění, zda zvolený polygon při průchodu jednotlivými body může být ucho či ne. Řešení tohoto problému bylo složitější, než jsem očekával. Platí, že pokud jakýkoliv z ostatních bodů právě rozkládaného polygonu je uvnitř potenciálního „uchového“ polygonu, polygon není uchem - znamená to totiž, že testovaný polygon není celý uvnitř polygonu, což vyvrací definici ucha. To se dá poměrně jednoduše vyřešit výpočtem obsahů trojúhelníků skládajících se vždy z jedné strany testovacího trojúhelníku a dvěma stranami vytvořenými spojením s bodem testovaným na toto kritérium vůči obsahu trojúhelníku vytvořeného testovaným polygonem - součet obsahů vytvořených ze stran a bodů se rovná obsahu celého testovaného polygonu za předpokladu, že je bod uvnitř tohoto trojúhelníku.[12] Pokud bych testoval polygony pouze pomocí tohoto kritéria, rozklad by proběhl tak jak má ve většině případů. To, že žádný bod není uvnitř testovaného polygonu však ještě neznamená, že je tento testovaný polygon celý uvnitř triangulovaného polygonu. To se stane pokud je jedna ze stran testovaného polygonu úplně mimo triangulovaný polygon - žádný bod tedy nemusí být uvnitř tohoto polygonu. Bylo tedy potřeba umět zjistit, kdy tomu tak je. Řešení, pro které jsem se rozhodl je následující :

- Najdi prostřední bod testované strany.
- Vyšli z tohoto bodu paprsek určitým směrem.
- Otestuj paprsek na kolize s ostatními stranami polygonu. (hledání průsečíku úseček, které je možno k tomuto zjištění využít, je podrobněji popsáno v kapitole 12).

- Pokud je počet kolizí sudý, bod je mimo polygon a tím pádem i testovaná strana, jinak je strana uvnitř polygonu.

Po spojení dvou výše zmíněných technik jde přesně určit, zda je určitý polygon ucho, či ne, a v závislosti na tom ho buď'to odříznout od zbytku polygonu, či otestovat po něm následující bod na zmíněná kritéria. Toto testování je vhodné provádět např. ve „while“ cyklu, či rekurzivně, s určitým ošetřením proti nekonečnému zacyklení. Já jsem se rozhodl pro rekurzivní funkci, ve které ošetřuji nekonečné zacyklení a následné přetečení zásobníku číslem, které se inkrementuje pokaždé, když se ve funkci nic nestane. Po určitém počtu zbytečných volání se z funkce vyskočí ven - tato situace by však neměla nastat. Rekurzivní funkce v každé své iteraci testuje a pokouší se postupně rozkládat daný polygon a ukládat oddělené polygony - např do listu. Jakmile je počet bodů 3, udělá se z těchto bodů poslední polygon, uloží se ke všem ostatním polygonům tvořící původní jednotný polygon, nyní však již rozdělený na množinu polygonů konvexních.



Obrázek 8: polygon po triangulaci - červenou barvou je zobrazeno jedno z uší

10 Umělá inteligence

10.1 Úvod

Ve hře kromě ostatních připojených hráčů nejsou žádní jiní nepřátelé. To mi poněkud zjednodušilo navrhování toho, jak „nadstavím“ třídy starající se o umělou inteligenci nad zbytek již implementovaného kódu. Rozhodl jsem se vnímat hráče ovládané umělou inteligencí stejně, jako hráče ovládané člověkem. Vlákno starající se o vykreslování světa, snímání klávesnice jsem vyměnil za vlákno, které svět nevykresluje a o stisknuté klávesy se v ní stará umělá inteligence, která se na základě světa kolem hráče jím ovládaným rozhoduje, jakou klávesu „stiskne“, či kam zamíří. Data jsou na straně serveru zpracována stejným způsobem, jako u ostatních hráčů (více viz kapitola o síťové architektuře) - hráči ovládaní umělou inteligencí (dále boti) tak nemají žádné výhody co se týče pohybu - speciálně tento bod přináší spoustu problému, se kterými jsem se musel poprat při návrhu i implementaci. Boti musí umět skákat přes překážky, střílet na hráče - pokud ho tedy uvidí, manévrovat v souboji, apod. První funkcionalitu kterou jsem se rozhodl implementovat a na ní vše postavit je určitý druh hledání cesty mapou - rozhodl jsem se pro A* algoritmus[14], který je jeden z nejznámějších algoritmů, především co se týče 2D her.

10.2 Hledání cesty grafem - A* algoritmus

10.2.1 Uzel

A* (A Star) algoritmus definuje uzel jako svou jednotku - obvykle čtvercovou kvůli jednoduchosti, používají se však i ostatní druhy polygonů. Každý uzel obsahuje následující informace :

- rodič - uzel může, ale nemusí mít rodiče. Rodič je uzel, který jde vnímat jako směr, odkud algoritmus do daného uzlu přišel.
- hodnota H (Heuristická hodnota) - hodnota určující vzdálenost daného uzlu od uzlu cílového. Dá se vypočítat mnoha způsoby, každý z nich vytváří svůj specifický typ pohybu.
- hodnota G (Cena pohybu) - tato hodnota je určena hodnotou G od rodiče daného uzlu + cenou cesty od rodiče k danému uzlu.
- hodnota F - kombinace hodnot H a G - používá se při výběru vhodných uzlů pro jednotlivé iterace algoritmu.

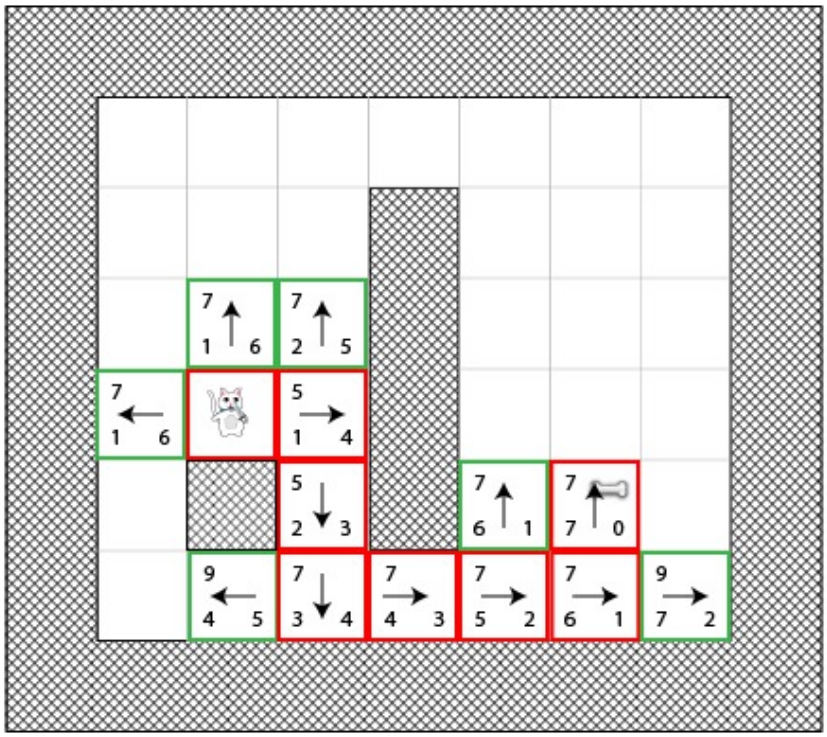
10.2.2 Otevřený/Uzavřený list

Uzly se obvykle vkládají do dvou listů : otevřeného a uzavřeného listu. V otevřeném listu se nacházejí všechny použitelné uzly v dané iteraci algoritmu, v uzavřeném listu jsou všechny již prozkoumané uzly v iteracích předchozích - tím je zamezeno navštívení

již prozkoumaného uzlu. Poté se odeberou či označí např. booleovským atributem uzly, které jsou ve světě v kolizi - to je vhodné předem vypočíst (za předpokladu že je svět statický).

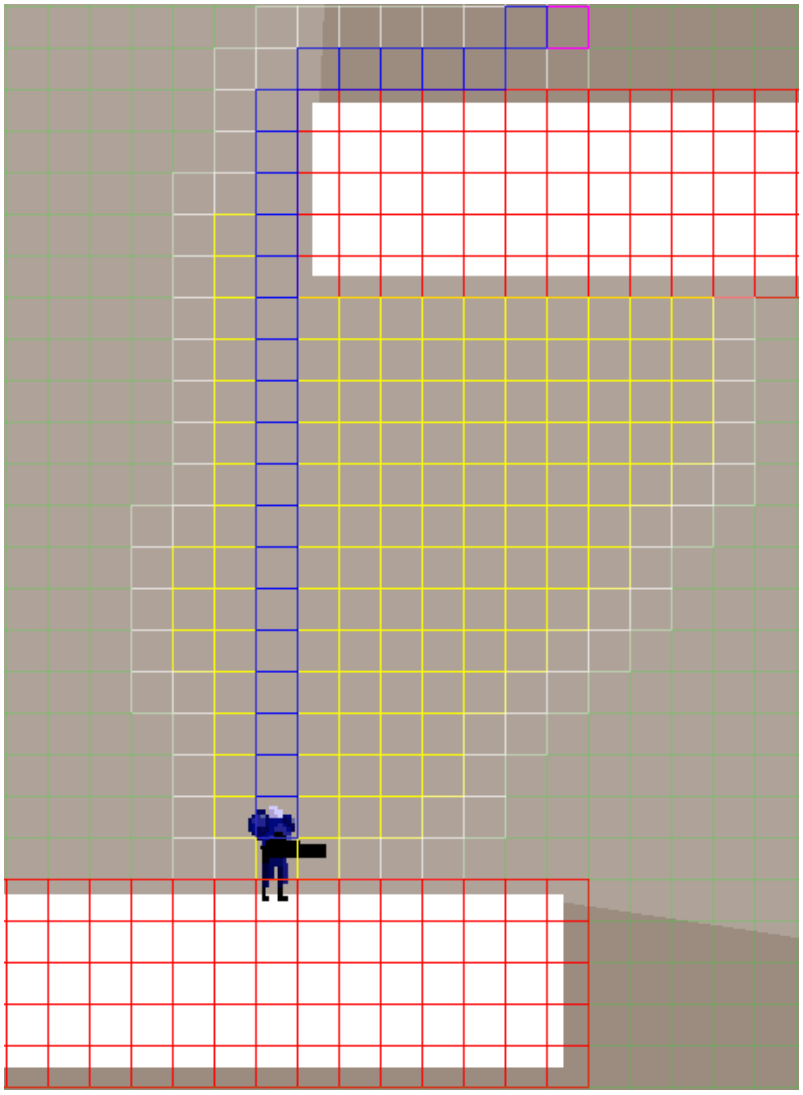
10.2.3 Popis algoritmu

Na začátku iterace algoritmu se zkoumají uzly v okruhu zvoleného centrálního uzlu (v první iteraci to je uzel počáteční) - vypočítají se jejich hodnoty H, G, a následně F - pouze však pokud tyto uzly nejsou kolizní a nejsou v uzavřeném listu. Všechny tyto uzly se vkládají do otevřeného listu, a jejich rodič je nastaven na centrální uzel. Centrální uzel je vložen do listu uzavřeného. Pro další iteraci se volí uzel s nejmenší hodnoutou F (který je použitelný) - v případě více stejných minimálních hodnot F je vhodné zvolit uzel s nižší hodnotou H. Iteruje se, dokud není nalezen uzel, který je na začátku iterování označen jako konečný. Jakmile je tento uzel nalezen, je funkce úspěšně ukončena vrácením listu uzlů tvořící nejkratší cestu od začátku k cíli (v rámci nastavených možností pohybu). Tento list je vytvořen zpětným návratem z konečného uzlu na uzel počáteční pomocí atributu rodič.



Obrázek 9: Hledání cesty grafem pomocí A* algoritmu - obrázek převzat z WWW : *http://www.raywenderlich.com/4946/introduction-to-a-pathfinding*

Na obrázku 9 je zobrazen výsledek postupu kočky, která hledá pomocí A* algoritmu cestu ke kosti. Kočka se přitom nemůže pohybovat diagonálně. Červenou barvou je znázorněna cesta. V jednotlivých uzlech se nachází hodnoty F (levý vrchní roh), G (levý spodní roh) a H (pravý spodní roh). Hodnota H je zde počítána Manhattanskou vzdáleností (horizontální vzdálenost + vertikální vzdálenost).



Obrázek 10: Zobrazení prohledávaných uzlů (žluté uzly) při hledání cesty (modré uzly) k cíli (růžový uzel)

10.2.4 Použití

Po implementaci tohoto algoritmu jsem musel vymyslet, jak se bude vypočítaná cesta používat. Jak bude bot vědět, kam se pohybovat? Pokud bot vidí hráče, vyřešil jsem to jednoduše - začne střílet, rychle se hýbat zleva doprava a občas vyskočí. Pro implementaci této funkcionality by prakticky nebylo nutné použít A* algoritmus (hráč nevidí ostatní hráče pokud mezi nimi existují překážky). Co ale dělat když bot žádného hráče nevidí? Řešení : bot si vybere jeden ze zajímavých bodů na mapě, a bude se na něj snažit dojít. Pokud se mu po cestě něco nestane (např. smrt), potom co dojde do daného bodu, vybere si další. Tyto body jsou rozmístěny po úrovni při její tvorbě tvůrcem mapy. Pokud by hra byla např. isometrická, nebo by hráči uměli lítat, jednoduše by se následovala cesta, která se vypočítá A* algoritmem. V mé hře však hráči lítat neumí, dále hráče ovlivňuje gravitace, a skákat umí jen do určité výšky. Protože jsem chtěl, aby byl editor map co nejjednodušší, jakákoliv úprava uzlů (např. označit určité uzly jako uzly „skokové“, aby bot věděl, že má skočit) byla vyloučena. Rozhodl jsem se trochu zaimprovizovat, a namísto rozšíření možností editoru map jsem rozšířil možnosti umělé inteligence. Rozšíření je následující : bot vypočítá cestu k cíli, nahlídne na N-tou pozici (např. pozice 3) a spočítá Y a X vzdálenost od zvoleného uzlu. Pokud je X vzdálenost minusová, půjde bot doleva, pokud bude plusová, půjde doprava. Protože bot vždy nahlíží na stejnou pozici v cestě k cíli, dá se pomocí Y vzdálenosti rozhodnout, zda je vhodné vyskočit. Stále však může nastat situace, že nebude moct vyskočit např. na plošinu, která je příliš vysoko (příklad této situace je vyobrazen na obrázku 10), a bude se o to pořád dokola pokoušet- proto jsem se do editoru map přidal možnost „neviditelných“ kolizních objektů, které jsou ve hře vnímány pouze při výpočtu A* cesty. Je tedy možné velice jednoduše zatarasit cesty „nesmyslné“ pro bota, a to s minimálním úsilím.

11 Pole viditelnosti - Vrhání paprsků

11.1 Úvod

Protože se hráči nevidí přes ostatní kolizní objekty, bylo vhodné tento fakt doplnit nějakou indikací, podle které se hráč může řídit - přesně tak vědět, kam vidí a kam ne, bez nutnosti hráčovy aproximace. Po internetové inspiraci[13] jsem se rozhodl vyřešit implementaci pole viditelnosti pomocí vrhání paprsků, kterému budu věnovat tuto kapitolu.

11.2 Popis algoritmu

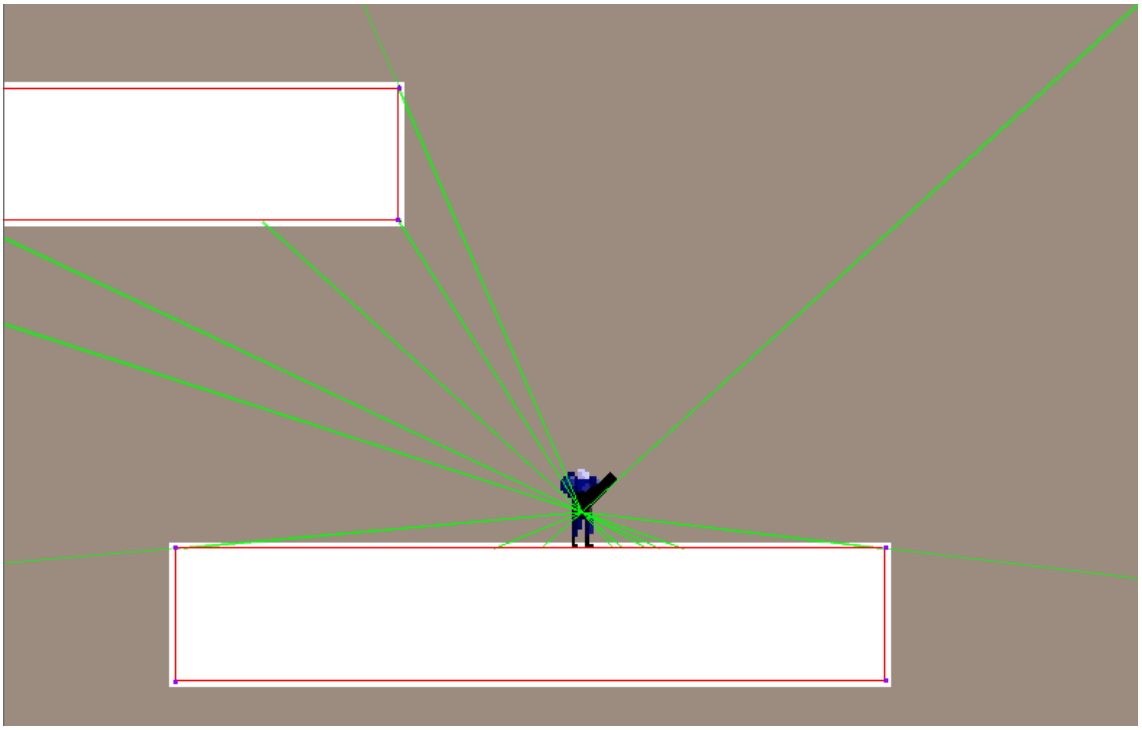
Algoritmus by se dal shrnout do následujících bodů :

- Vyšli paprsek ke každému viditelnému bodu.
- Ke každému z těchto bodu vyšli další dva paprsky, tentokrát s určitou odchylkou $+/-$: odchylka by měla být malá, okem neviditelná (např. setiny stupňů).
- Najdi nejbližší průsečík(hledání průsečíku úseček je podrobněji popsáno v kapitole 12) všech těchto paprsků se světem (nejlépe pouze v nějakém rozsahu pro zrychlení algoritmu). Testují se pouze obrysy kolizních objektů, nikoliv všech triangulovaných polygonů, který je tvoří - strany uvnitř polygonu by byly zbytečně testovány (paprsky po otestování jsou zobrazeny na obrázku 11).
- Seřaď průsečíky podle úhlu, který svírá úsečka tvořená průsečíkem a středem hráče s např. osou X.
- Vytvoř polygony skládající se vždy ze dvou sousedních průsečíků nalezených v předešlých bodech a prostředním bodem hráče.

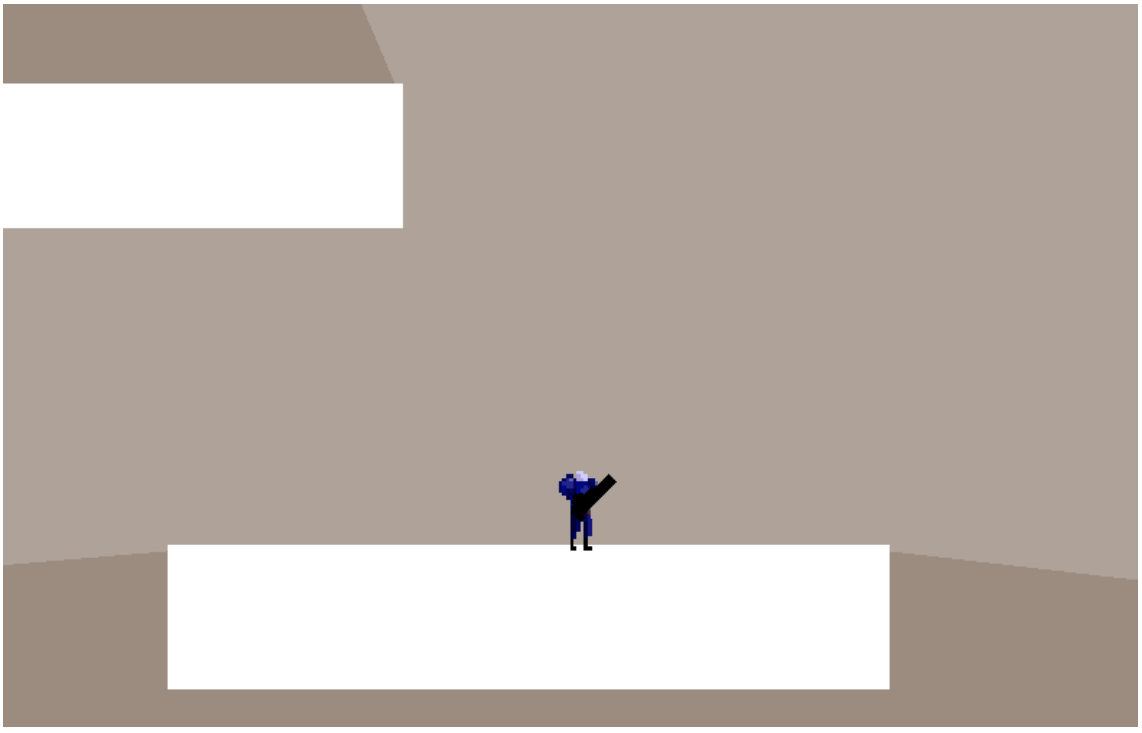
Důležitým poznatkem, který výše zmíněný internetový článek [13] nezmiňuje je fakt, že musí být kolem viditelného obrazu hráčem vytvořen box, který slouží jako „zachycovač“ paprsků, které nejsou s ničím jiným ve světě v kolizi - každý paprsek musí mít průsečík ve světě, aby tato metoda fungovala tak, jak má. Zároveň je nutné vysílat paprsky vůči všem čtyřem bodům tvořícím tento box - to způsobí vytvoření polygonů na místech, kde se nevyskytují žádné ostatní kolizní body vůči kterým vysílat paprsky.

11.3 Použití ve hře

Poté, co algoritmus úspěšně proběhne, vykreslí se vytvořené polygony na obrazovku např. s určitým vyplněním s nízkým prvkem alfa (zobrazeno na obrázku 12). Samotný výsledek algoritmu se logicky nestará o „skrytí“ objektů, které je nutné skrýt - v zásadě by mohl, avšak to by bylo vysoce neefektivní a zbytečně složité (zvláště co se týče umělé inteligence). Zjištění viditelnosti se provádí opět pomocí vysílání paprsků vůči pohyblivým entitám na obrazovce. Pokud úsečka vytvořena mezi např. hlavou hráče a prostředkem těla nepřítele není nikde přerušena, nepřítel je vidět, jinak ne.



Obrázek 11: Zobrazení vyslaných paprsků po testování na průsečíky



Obrázek 12: Vykreslení polygonů s určitým podbarvením

12 Nalezení průsečíku dvou úseček

Průsečík dvou úseček se dá nalézt různými způsoby. Zvolil jsem ten, který mi byl nejbližší - řešení pomocí soustavy lineárních rovnic přímek. Každá přímka lze zapsat následující rovnicí :

$$y = ax + b$$

Proměnná a značí sklon přímky. Podle této hodnoty lze také zjistit, zda jsou přímky rovnoběžné (když se hodnoty sklonu a u obou přímek rovnají). Lze vypočítat následujícím vzorcem, kde proměnná $pBod$ značí počáteční bod úsečky a proměnná $kBod$ bod konečný :

$$a = \frac{kBod_y - pBod_y}{kBodx - pBodx}$$

Pro proměnnou b platí, že přímka musí protnout osu Y v bodě (0,b), a lze získat substitucí hodnot x a y za body, které již na přímce známe - lze tedy použít počátek i konec zadané úsečky. Za předpokladu, že jsou známy hodnoty a a b obou přímek, lze od sebe rovnice přímek s těmito dosazenými hodnotami odečíst, a získat tím hodnotu x průsečíku, která se následně dosadí do jedné z předešle použitých rovnic, z čehož se vypočte hodnota y průsečíku. Tento průsečík však ještě neznačí hledaný průsečík dvou úseček, nýbrž průsečík dvou přímek. To lze vyřešit testem na to, jestli je bod úsečky „mezi“ body přímek - pokud ano, je nalezen průsečík dvou úseček.

12.1 Převod do kódu

I když se jedná o řešení pomocí soustavy rovnic, žádné soustavy rovnic se v kódu neobjevují - Java nedokáže sama o sobě řešit soustavy rovnic o několika neznámých. Platí však, že :

$$b1 - b2 = -(a1 \cdot x) + (a2 \cdot x)$$

Tohoto faktu jsem využil k získání hodnoty x průsečíku : ta se získá pokud se $b1 - b2$ vydělí číslem $(-a1 + a2)$, a následně se může vložit do kterékoliv z rovnic přímek ze zadaných úseček pro získání hodnoty y průsečíku. Až po převodu do kódu jsem si uvědomil další důležitý fakt nutný ošetření : Pokud je přímka rovnoběžná s osou Y v Eukleidovském prostoru, vychází nekonečný sklon, což je problém při následném výpočtu hodnoty b. Pokud taková situace nastane, dá se toho však využít, protože bod x dané úsečky se v tu chvíli již nemusí počítat - nikdy se totiž nezmění od bodu počátečního či koncového.

13 Závěr

Ač jsem se v práci snažil popsat pouze ty nejdůležitější problémy řešené při programování hry, bylo jich příliš mnoho na to, abych je popsal všechny, či při daném popisu zacházel do přílišných detailů. I přesto věřím, že tato práce může být přínosná jako vzor pro implementaci probírané funkcionality - zejména proto, že mi při programování samotnému články s podobnou stručností vyhovují. Při tvorbě hry jsem se dozvěděl spoustu zajímavých a užitečných technik, kterými se řídí spousta jak nezávislých, tak AAA her. Zadání práce bylo zároveň velmi volné, což mi umožnilo volit funkcionalitu aplikace přesně podle svých představ. V tom mi také velmi pomohl můj vedoucí práce, se kterým jsem své volby pravidelně konzultoval a inspiroval se jeho nápady při implementaci velké části herní funkcionality. Protože jsem pracoval na relativně nízké aplikační úrovni - vzhledem k volbě knihoven - pomohlo mi to pochopit některé z problémů více do hloubky a také mít větší kontrolu nad funkcionalitou aplikace než při použití jednoho z herních „enginů“. Co se týče hry samotné - v době psaní této práce je ve fázi hratelného prototypu poukazujícího na všechnu funkcionalitu popisovanou v této práci. Rád bych v blízké budoucnosti hru dokončil a v případě dokončení nyní vyvíjeného portálu her ji do něj zařadil.

14 Reference

[1] *Performance - OpenGL.org* [online]. poslední revize 13. května 2014 [cit. 28.4.2015]. Dostupné z WWW: <https://www.opengl.org/wiki/Performance>.

[2] *opengl-tutorial.org | Tutorials for modern OpenGL(3.3+)* [online]. poslední revize 2. února 2015 [cit. 28.4.2015]. Dostupné z WWW: <http://www.opengl-tutorial.org/>.

[3] *Slick2D | 2D Java Game Library* [online]. poslední revize 10. ledna 2015 [cit. 28.4.2015]. Dostupné z WWW: <http://slick.ninjacave.com/>.

[4] *EsotericSoftware/kryo - GitHub* [online]. 2015 [cit. 19.4.2015]. Dostupné z WWW: <https://github.com/EsotericSoftware/kryo>.

[5] *libgdx* [online]. 2015 [cit. 19.4.2015]. Dostupné z WWW: <http://libgdx.badlogicgames.com/>.

[6] *lwjgl* [online]. 2015 [cit. 19.4.2015]. Dostupné z WWW: <http://www.lwjgl.org/>.

[7] GAMBETTA, Gabriel. *Gabriel Gambetta - Fast-Paced Multiplayer* [online]. 2015 [cit. 19.4.2015]. Dostupné z WWW: <http://www.gabrielgambetta.com/fast_paced_multiplayer.html>.

[8] *Source Multiplayer Networking - Valve Developer Community* [online]. Poslední revize 23. června 2014 [cit. 19.4.2015]. Dostupné z WWW: <https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking>.

[9] COZIC, Laurent. *2D Polygon Collision Detection - CodeProject* [online]. 13. září 2006, poslední revize 20. září 2006 [cit. 19.4.2015]. Dostupné z WWW: <http://www.codeproject.com/Articles/15573/D-Polygon-Collision-Detection>.

[10] BITTLE, William *SAT (Separating Axis Theorem) | dyn4j* [online]. 1. ledna 2010 [cit. 19.4.2015]. Dostupné z WWW: <http://www.dyn4j.org/2010/01/sat/>.

[11] *GameDev math recipes: Collision detection using an axis-aligned bounding box* [online]. 26. listopadu 2012 [cit. 19.4.2015]. Dostupné z WWW: <http://www.gamefromscratch.com/post/2012/11/26/GameDev-math-recipes-Collision-detection-using-an-axis-aligned-bounding-box.aspx>.

[12] *Check whether a given point lies inside a triangle or not - GeeksforGeeks* [online]. 2012 [cit. 19.4.2015]. Dostupné z WWW: <http://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/>.

[13] CASE, Nick. *SIGHT & LIGHT - how to create 2D visibility/shadow effects for your game* [online].[cit. 20.4.2015]. Dostupné z WWW: <http://ncase.me/sight-and-light/>.

[14] *Introduction to A** [online]. 2010 [cit. 22.4.2015]. Dostupné z WWW: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>.

[15] *Java Persistence Performance : Optimizing Java Serialization* [online]. 13. srpna 2013 [cit. 22.4.2015]. Dostupné z WWW: <http://java-persistence-performance.blogspot.cz/2013/08/optimizing-java-serialization-java-vs.html>.

[16] Meisters, G. H., *Polygons have ears.*. American Mathematical Monthly 82, 1975. 648–651.

15 Přílohy

Přílohy na CD/DVD :

Adresář	Obsah
Hra	spustitelná hra a mapy
Kód	zdrojový kód aplikace
Dokumentace	PDF s uživatelskou dokumentací aplikace
Práce	práce v elektronické podobě ve formátu PDF